BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

Teuvo Kohonen

Content-Addressable Memories

With 123 Figures

Springer-Verlag Berlin Heidelberg New York 1980

Springer Series in Information Sciences

Volume 1 **Content-Addressable Memories**
By T. Kohonen

Volume 2 **Fast Fourier-Transform and Convolution Algorithms**
By H. Nussbaumer

Professor Teuvo Kohonen

Department of Technical Physics, Helsinki University of Technology
SF-02150 Espoo 15, Finland

Series Editors:

Professor King Sun Fu
Professor Thomas S. Huang

School of Electrical Engineering, Purdue University
West Lafayette, IN 47907, USA

Professor Dr. Manfred R. Schroeder

Drittes Physikalisches Institut, Universität Göttingen
Bürgerstrasse 42–44, D-3400 Göttingen, Fed. Rep. of Germany

ISBN 3-540-09823-2 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-09823-2 Springer-Verlag New York Heidelberg Berlin

Library of Congress Cataloging in Publication Data. Kohonen, Teuvo. Content-addressable memories. (Springer series in information sciences; v. 1). Bibliography: p. Includes index. 1. Associative storage. 2. Information storage and retrieval systems. I. Title. II. Series. TK7895.M4K63 621.3819'533 79-24499

Printed in Germany

Offset printing and bookbinding: Brühlsche Universitätsdruckerei, Gießen
2153/3130-543210

control, in the way described below in this section. Content-addressable memories which have provisions for the within-memory masking are named *functional memories* (FMs); for the implementation of logic functions, they usually include some auxiliary circuits for the collection of responses from the matching words, too. These constructs, as introduced by FLINDERS et al. [3.59], as well as GARDNER [3.60,61], have the same objective as the microprogram memories, i.e., to make the detailed machine operations and the control logic programmable. The FMs in fact allow the representation of truth tables in a highly compressed form.

It turns out that for the representation of the three symbols 0, 1, and Ø, at least two binary storage elements per bit cell are needed. Pairs of bit values can be assigned in four different ways to three symbols, e.g., 0 ⇔ (1,0), 1 ⇔ (0,1), and Ø ⇔ (0,0). In a hardware design, two bistable circuits together with some comparison logic are needed for every bit cell.

3.7.1 The Logic of the Bit Cell in the FM

One schematic implementation of the FM bit cell, somewhat similar to that of the CAM bit cell shown in Fig. 3.3 but without addressed readout, is presented below in Fig. 3.22.

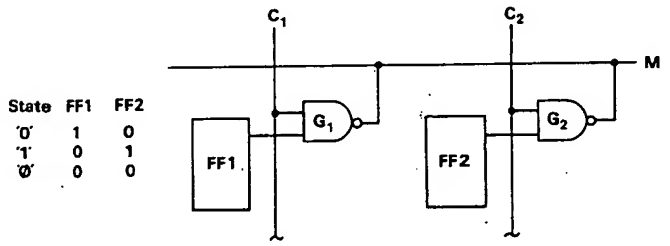

Fig. 3.22. Content-addressable reading function of the FM bit cell

The central idea applied in this circuit is the following. If no masking in the search argument is used, in searching for '0' one has $C_1 = 0$, $C_2 = 1$, and in searching for '1' there is $C_1 = 1$, $C_2 = 0$. Assume now that both flip-flops are in the 0 state corresponding to value 'Ø'; since there are only gates G_1 and G_2 connected to the M line, contrary to the double gates G_6 and

G_7 of Fig. 3.3, no mismatching signals can be caused on the M at any value combination of C_1 and C_2 whatsoever.

Only a few electronic implementations of the above principle have been suggested, although all basic circuit constructs of the CAMs are in principle amenable to the FM.

3.7.2 Functional Memory 1

The first of the FM types [3.19] is intended only to implement disjunctive forms of Boolean functions. They all can be expressed in a general form

$$F = \bigvee_{P} (A_1^{p_1} \wedge A_2^{p_2} \wedge \ldots \wedge A_n^{p_n}) \qquad (p_1,p_2,\ldots,p_n) \in P \tag{3.17}$$

where the superscript p_i, $i = 1,2,\ldots n$ is assumed to attain one of the values 0, $\emptyset$, and 1, P is a set of combinations of the superscripts, and $A_i^{p_i}$ is an operational notation with the meaning

$$A_i^0 = \overline{A}_i,\ A_i^{\emptyset} = 1, \text{ and } A_i^1 = A_i .$$

The expressions $\overline{A}_i$ and A_i are named *literals* corresponding to an independent variable A_i.

The notations expressed in (3.17) have now a very close relationship to the so-called *compressed truth table* which is obtained, e.g., by the well-known *Quine-McCluskey method* (cf, e.g., [3.62]). The following example, Table 3.5, shows a usual and a compressed truth table, respectively.

Table 3.5. An example of truth tables

Usual				Compressed			
A	B	C	F	A	B	C	F
0	0	0	0	0	1	$\emptyset$	1
0	0	1	0	1	$\emptyset$	0	1
0	1	0	1				
0	1	1	1				
1	0	0	1				
1	0	1	0				
1	1	0	1				
1	1	1	0				

In the combination of rows, the Ø signifies a "don't care". It is also possible to combine rows including Ø's in the same positions whereby new rows with more Ø's are obtained.

When comparing the compressed table with the Boolean expression (3.17) in which the same simplifications have been made as those which have led to the compressed table, then *every row in the truth table corresponds to one product term in (3.17), and the row is identical with the word* $(p_1,p_2,\ldots,p_n)$.

Obviously, if the prevailing value combination of the independent logic variables is compared with all rows of the table, the matching of a row in all specified bit positions (i.e., when no attention is paid to the Ø's) can only occur at the rows shown. A comparison of this type differs from the masked comparison operations discussed with the CAMs in that Ø corresponds to a mask which is set *within the table* and not in the search argument.

It is now possible to implement the compressed truth table using FM hardware, by storing the left half of the table in a special memory where content-addressable comparisons are performed. The memory cell, however, must then be able to attain one of three possible state values denoted by 0, 1, and Ø, respectively, and the bit comparison shall be masked out at Ø.

The FM is also suitable for simultaneous representation of several Boolean functions. The FM then consists of two parts, named the Input Table and the Output Table, respectively. As an illustration, we shall consider the incrementation of 8421-coded numbers mod 16, according to [3.18]. The simplified Boolean functions for the result bits (R_i) together with the carry (C), for all combinations of source bits (S_i) are given as follows:

$$
\begin{aligned}
C &= S_3 \wedge S_2 \wedge S_1 \wedge S_0 \\
R_3 &= (\bar{S}_3 \wedge S_2 \wedge S_1 \wedge S_0) \vee (S_3 \wedge \bar{S}_2) \vee (S_3 \wedge \bar{S}_1) \vee (S_3 \wedge \bar{S}_0) \\
R_2 &= (\bar{S}_2 \wedge S_1 \wedge S_0) \vee (S_2 \wedge \bar{S}_1) \vee (S_2 \wedge \bar{S}_0) \\
R_1 &= (\bar{S}_1 \wedge S_0) \vee (S_1 \wedge \bar{S}_0) \\
R_0 &= \bar{S}_0 \quad .
\end{aligned}
\tag{3.18}
$$

The combined truth table is shown in Table 3.6. The Ø's of the Input Table, and the 0's of the Output Table are indicated by blanks, and this convention shall be followed throughout the rest of this section.

In the hardware implementation of the combined truth table, the Input Table has a similar FM counterpart as that described above, with MATCH signals obtained as outputs at every word line (row). Several logic sums are formed by OR circuits, one for every Boolean function. The inputs to these OR cir-

Table 3.6

	Input Table				Output Table				
Row	S_3	S_2	S_1	S_0	C	R_3	R_2	R_1	R_0
1				0					1
2			0	1				1	
3			1	0				1	
4		0	1	1			1		
5		1	0				1		
6		1		0			1		
7	0	1	1	1		1			
8	1	0				1			
9	1		0			1			
10	1			0		1			
11	1	1	1	1	1				

cuits are taken from those word lines which correspond to the bit value 1 in the columns of the Output Table.

It may be recalled that the main objective in the introduction of functional memories was the implementation of logic operations by programming. Accordingly, any specified hard-wired operations such as that described above in which word lines were connected to OR circuits according to the functions to be implemented, should not be allowable. The Output Table can now be made *programmable* by providing every location in it, i.e., every crossing of the word lines and columns by a usual flip-flop which can be read by the word line signal, and connecting the output circuits of all flip-flops of one column by a Wired-OR function. The value 1 is written into all flip-flops in which the Output Table has them, and so the vertical output line at every column will receive a resultant signal which corresponds to the hard-wired operation diescribed above. We shall revert to a similar programmed output operation with Functional Memory 2 below.

Search-Next-Read. Table 3.6 can further be compressed by introducing the Search-Next-Read function, as named by FLINDERS et al. The above example, due to its special properties, may yield a rather optimistic view of the applicability of this method, but similar cases may occur rather often. The central idea is that some input and output rows, not necessarily adjacent ones, may resemble each other. For example, in this example, input row 2 has a 1 in the same position as output row 1 has it; if a read operation were

possible in the *Input Table*, output row 1 could be represented by input row 2. If we would now make the convention that after searching for input rows, the *next* input rows are automatically read, output row 2 could be deleted. Similarly, the output rows 2, 3, 4, 5, 7, 8, and 9 could be deleted, because they have 1's in the same position as the input rows 3, 4, 5, 6, 7, 9, and 11, respectively, have them. (Notice that rows can be easily *reordered* in order to represent as many output rows by next input rows as possible.) A separate problem arises with output rows 6, 10, and 11, which cannot be represented by the next input rows. The solution is that these left-over output rows are added behind the corresponding input rows in the table, but now every row is provided with an additional *tag bit*; for words not allowed to occur as search arguments, as for the three ones mentioned last, this tag bit is 1, and it is 0 for the rest of the words. The source word, the search argument, now has an extra bit in this position and its value is 0 during searching; during reading it is 1. The words provided with a tag 1, therefore, cannot match with any search argument during searching, and do not interfere. During reading, the next to the searched word is always read. Table 3.7 shows the more compressed table.

NOTE

Table 3.7

Row	Tag	C	S_3 R_3	S_2 R_2	S_1 R_1	S_0 R_0
1	0					0
2	0				0	1
3	0				1	0
4	0			0	1	1
5	0			1	0	
6, input	0			1		0
6, output	1			1		
7	0		0	1	1	1
8	0		1	0		
9	0		1		0	
10, input	0		1			0
10, output	1		1			
11, input	0		1	1	1	1
11, output	1	1				

For instance, if the number obtained by incrementing 4, (i.e., 5) had to be generated by the functional memory, the search argument would be 000100. This matches with rows 1, 5, and 6 (input), and, therefore, rows 2, 6 (input), and 6 (output) are read out. The ORing of the respective bits yields 100101. By neglecting the leading 1, the tag, the result is 00101 which represents 5.

The tag concept can be developed further. By using different combinations of tags for different sets of rows, the same table can be used to represent different functions, just by providing the search argument by a function code which must match with particular tags in the searching operation. For further suggestions, see [3.59].

3.7.3 Functional Memory 2

The previously discussed functional memory was intended for a direct implementation of two-level logic, i.e., for disjunctive Boolean forms. With a minor amount of additional circuitry, certain multilevel logic expressions can advantageously be represented by a functional memory which thus can further be compressed. The first-level operation in these expressions is always AND, and the second-level operation is OR. The logic operation on the third level is EXOR (EXclusive OR), and the fourth level implements the ANDNOT function. These particular operations on different levels were selected for this implementation because it will be easy to transform truth tables into such expressions, as will be demonstrated below. In particular, the EXOR and ANDNOT forms can be found easily, e.g., by direct inspection of Karnaugh maps as illustrated by a couple of examples.

Let us first consider the Karnaugh map shown in Fig. 3.23a. The intersectional area of the two terms shown by dotted lines cannot be represented by their Boolean sum, but EXOR is a function which is directly suited for this purpose: the EXOR of these terms has zeros in the intersection.

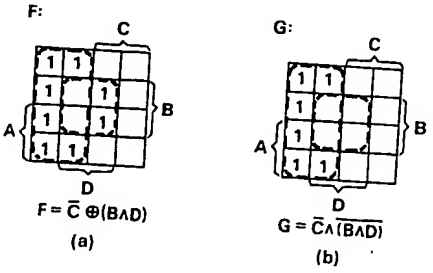

Fig. 3.23a,b. Finding terms for Functional Memory 2: a) for the EXOR function b) for the ANDNOT function (see text)